TAMPERE UNIVERSITY OF TECHNOLOGY

CARLOS PÉREZ BLANCO

NOSQL DATABASES IN CROSS-PLATFORM DEVELOPMENT

Master of Science Thesis

Examiner: Tommi Mikkonen
Examiner and topic approved by the
Faculty Council of Computing and
Electrical Engineering on 08.11.2013

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
PÉREZ BLANCO, CARLOS: NoSQL databases in cross-platform development
Master of Science Thesis, 50 pages, 1 appendix page
December 2013
Major: Software Systems
Examiner: Tommi Mikkonen
Keywords: nosql, couchdb, cross-platform development, android, web applications

Wellness coaching is nowadays a business on the rise. People are more aware of their health and would like to keep track of and improve their habits in the simplest and effortless possible way. Coaches, on the other hand, need tools that help them reach their full potential.

Movendos Oy was established in late 2012 with the aim of providing effective tools for health coaching. Before a commercial product could be released to the market, there was a need for piloting with selected customers using a functional prototype that was usable in all kinds of devices, from desktop computers to smartphones.

In this thesis such a prototype will be designed and implemented using different technologies. The use of NoSQL databases as a storage solution is getting popularity and fits the purpose of the system. Developing a hybrid application for Android smartphones adds value and flexibility for the user, getting the most out of the system and device.

This thesis proves how NoSQL databases can be a suitable solution for such an application, how hybrid applications can be the choice over web or native development for mobile devices, and how the prototype was professionally used by Movendos Oy's partners successfully. This success allowed the company to continue working on the development of the first commercial version, which has already been released and taken into use by several partners.

To my family, for supporting me always, but specially when being abroad.

To all my colleagues at Movendos, for giving me the opportunity to work on this project and write my thesis about it, and specially Stefan Bäggstrom for his guidance.

To my examiner Tommi Mikkonen for his inestimable help, positive attitude and understanding during this period.

To everyone I have met during these years studying at Tampere University of Technology.

CONTENTS

1. INTRODUCTION

1.1 Motivation

Health and wellness have become nowadays a trendy topic in modern society. The business of personal health coaching is getting bigger and bigger, but it is still under development and lacking tools that would help professional coaches get the most out of their business. As an example, communication with their customers usually takes place face-to-face and with several weeks in between meetings. Also keeping track of progress becomes a difficult task without the proper tools, both for customers and coaches.

Mobile devices are also part of people's daily lives. Development for this kind of devices has skyrocketed in the last few years. Applications developed specifically for tablets and smartphones have become an important feature for many businesses and provide a simple and time effective way for their users to consume and produce content. There are different approaches when deciding how to develop a mobile application, and this decision has a major impact on the final results. With most of the content being *in the cloud*, the subject of dealing with those situations where there is no active network connection is another issue to consider.

Finally, NoSQL databases' use in the industry has grown exponentially over the last few years and has posed a direct competition for traditional relational databases. Their use can give several benefits, such as a greater flexibility of the data model and an easier to scale system when dealing with great amounts of data.

1.2 Objectives

This thesis achieves four different goals. The first goal is to present the reader to the concepts of native, hybrid and web applications. Secondly, the reader will be shown the problem of offline functionality on mobile applications. Third, NoSQL databases will be introduced, along with some of the relevant concepts associated with their use. Finally, the fourth and main goal of this thesis is to design a fully functional prototype application for Movendos Oy, in order for the company to be able to run pilots with potential partners, applying in a suitable manner the concepts and techniques previously presented.

The developed prototype will consist of an application that is usable in desktop

computers, tablets and mobile devices. This application needs to meet essential performance requirements and be easy to learn and use, as target users may not be computer experienced.

In addition, the system should also be usable through an Android hybrid application that will support certain offline functionality, for which local resources are required in the device. One way to achieve this is using replication to the device so that all transactions are local. The use of NoSQL databases like CouchDB and TouchDB suits well this approach, while most things can be achieved with JavaScript, easing the learning curve.

1.3 Structure

This thesis is divided into 6 chapters:

Chapter 1, "Introduction", includes an introduction to the document, describing its motivation and objectives. Chapters 2, "Mobile applications and offline functionality", and 3, "NoSQL databases" define the basic concepts the reader should be familiar with and give an insight into the current status of fields and techniques relevant for the thesis' topic. Chapter 4, "Case study: Movendos prototype application", presents the case study itself, the developing of a prototype application for Movendos Oy. The design and implementation process is described and the software is detailed. Chapter 5, "Evaluation", evaluates the results of the thesis work and exposes the conclusions.

2. MOBILE APPLICATIONS AND OFFLINE FUNCTIONALITY

2.1 Native, web and hybrid apps

In the past few years, development for mobile devices has grown exponentially, especially since the launch of the first smartphone generation [1]. When Apple launched the first iPhone in the middle of 2007, it was not possible for third-party developers to produce software for the device, at least not through any legal channel. It was not until one year later in July 2008, when Apple launched their own distribution channels for third-party applications, called the App Store. Other companies have since followed the same path, such as Google and Google Play, BlackBerry and BlackBerry World or Nokia and Ovi Store to cite a few, while smartphone and recently tablet usage has also grown quickly.

Nowadays, when smartphones have a much higher penetration rate in the market, developing a mobile application has also become much easier and widespread, as there is a great number of development tools that make the process simpler. Usually companies release publicly their Software Development Kit (SDK) which, together with an Integrated Development Environment (IDE) makes it easy for developers to write applications for their devices.

However, the vast number of platforms available also poses a potential downside. In most cases, different platforms require different tools and their applications are written in different programming languages. An application written for Android will never run on any Apple device, for example. This is certainly a burden for developers, who aim to cover as much market as possible, and rises an obvious question: *"How can I develop an application that runs on multiple devices while spending the smallest amount of resources" "Do I need to rewrite the application for every new platform I want it to be released to?"*

Luckily, there are three different types of mobile applications with contrasting principles and characteristics, which offer a suitable solution for each specific case's needs. In fact, each one of them has many implications that will determine a big number of the application's features, from performance to the way it can be monetized. The three types of mobile applications are native, web and hybrid applications, and we will describe and analyze them in the following paragraphs.

2.1.1 Native applications

Native applications are, in terms of look, feel and overall performance, the best possible approach. They are designed and written specifically for a certain platform or device, using the development tools that companies provide, in one of the languages the platform supports. Among its main benefits we can cite the following:

- Full hardware feature of the device the application is designed for. This means it will be able to access all the built-in components, such as notifications, camera, microphone, geolocation through Global Positioning System (GPS), accelerometer or the device's address book.
- Because they make native use of the platform's capabilities, the application benefits from the fastest and most fluid graphics. Native User Interface (UI) components are readily available, making the application look and feel better. In general terms, the performance is the best, which is most important for the end user.
- Being able to sell the application directly on the manufacturer's app store, in addition to adding advertising and sponsorship.

On the other hand, developing a native application also has certain disadvantages:

- The need to rewrite the application if we want to launch it for a different platform. Sometimes one can take little benefit from using the same language (both Android and Blackberry use Java), but some other times the application needs to be written from scratch, which leads to development costs being higher than other alternatives. However, the recent proliferation of commercial cross-platform tools and SDKs such as Qt [2] and Apache Cordova [3] that generate code compatible among multiple platforms has lately made this a smaller issue than it used to be.
- Related to the previous, the cost of maintaining and updating all the different versions of the application is also higher than that of the other alternatives. Native applications usually requires more projects and resources, and scales up exponentially.
- Some manufactures, like Apple, may apply fees, approval policies and review the application in order to be published to their marketplace, thus making the publishing process tedious and time consuming.

2.1.2 Web applications

The opposing philosophy is represented by web applications. These are applications which are stored in a server and thus can be distributed freely without the user having to download any special package, making cross platform compatibility one of their main advantages, among others:

- Since the application runs on the device's Web browser, it works on any platform, ensuring compatibility among all of them. This implies also being able to reach users using less common platforms without any extra requirements. However, this can also turn a disadvantage as not all features are always available.
- The use of HTML [4], CSS [5] and JavaScript [**?**] for most of the platforms makes the development process simpler and reduces its costs.
- Since they cannot be deployed to the platform's marketplace, there is no need for any official revision from the manufacturer, so updates reach every user automatically and at the same time.

If we look at the disadvantages, we can cite the following:

- Access to the native components of the device is restricted or simply not possible, such as the native file system, camera or geolocation. This could change in the near future as we will discuss later, but for the time being one should assume it will not be possible to use them.
- Web applications tend to be less responsive than native ones, since they cannot benefit of the native UI components.
- The application cannot be distributed through the device's marketplace, so developers need to make an additional effort to sell the application. However, monetization is still possible, mainly through advertising and sponsorship, but also by using subscription or *pay-per-use* mechanisms.
- Despite theoretically being usable in any platform, compatibility among different Web browsers cannot be guaranteed. This is specially true for newest HyperText Markup Language (HTML)5 features, which only work on selected browsers.

2.1.3 Hybrid applications

As we can see, both of these approaches perform best when the other lacks, so ideally one would aim to merge the best features of each of them. The third alternative

is called hybrid applications. These applications combine the best of both previous approaches by offering a mobile optimized web application wrapped inside a native application package.

This native layer is the responsible for providing access to the built-in components of the device, meaning that camera, GPS, microphone and others can be accessed just the way you would do with a native application, so the applications are richer since they have several means to interact with the user and its environment.

In addition, as mentioned before the core of the application is still a simple web application, so targeting several platforms is easier from the architecture point of view.

Besides, all the native visual style is also available, so it is common to combine both web and native elements in the applications. Usually complex and flexible views are implemented using HTML and wrapped within a native web view, while features that are not subject to many changes such as menus can use the native components.

As a result, applications not only are responsive and have access to any native component, but also development costs are reduced and the resulting package can be distributed through the official channels.

On the other hand, hybrid applications are not always the ideal solution, since they add extra complexity and they may not apply in certain scenarios where the web application cannot simply be wrapped into a native package.

Figure 2.1 below summarizes this information by comparing visually the main characteristics of native, hybrid and web applications.

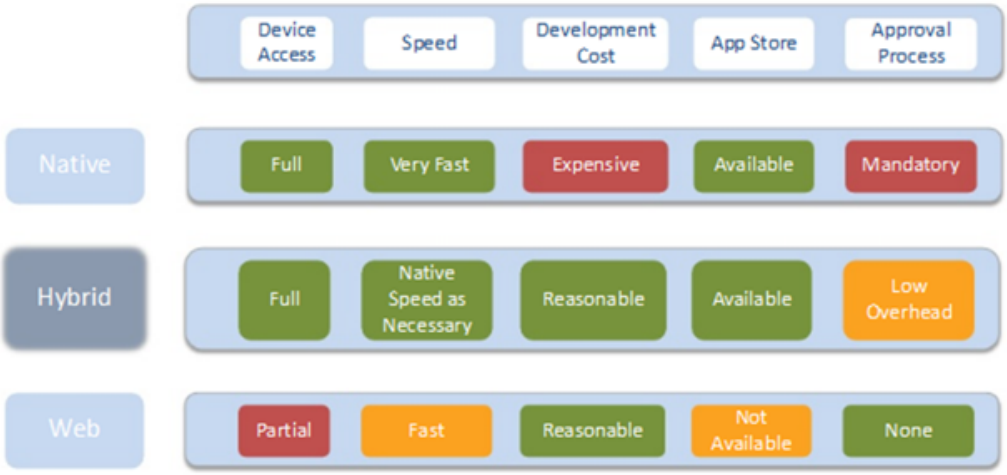


Figure 2.1: Comparison of main features among native, hybrid and web applications. [7]

While these are the most important and relevant aspects that have to be considered when designing a mobile application, there are many other complex factors

that also affect the decision. As cited in [8], external factors such as demographics also affect the way the users are going to use an application, given their country of residence, age or even gender.

According to the same source, the type of content, where the application is used, whether it can be used offline or the delay of the contents are other factors to take into account when designing an application. For instance, the fact that native applications support notifications makes them an interesting option to consider when data is more *alive*, as the user can get notified instantly. Those applications that need to support heavy calculations also should perform better when being native.

From this analysis one could conclude that native applications are richer when it comes to features, they can interact with native functionality and perform better when compared to web applications, whereas these are easier and cheaper to develop if we are targeting several platforms. Hybrid applications combine these features but also carry on their inconveniences. The user is forced to install an application which still performs poorly when compared to its Web counterpart, since the UI is executed from a web context and access to native functionality must get through an intermediate layer. Hence, the choice depends greatly on each specific scenario.

It is worth noting that this scenario might change in the near future as the World Wide Web (WWW) evolves and gradually moves towards the use of the new HTML5 standard [9].

This new revision of the standard by World Wide Web Consortium (W3C) specifies new Application Programming Interfaces (APIs) that enable file manipulation, Web storage, media capture and geolocation among others, as can be seen in Figure 2.2 below.

At the time of writing, some of the most widespread mobile and desktop web browsers like Chrome, Firefox, Safari and Opera already support part of these features, which will probably soon become the new standard and improve web applications' capabilities in terms of device access a user experience, effectively bringing the web experience to a level much closer to that of a native application.

2.2 Offline functionality in mobile apps

Offline capability is an important factor that needs to be taken into account when designing a mobile application. Nowadays, data connection and wireless networks have become widespread in developed countries and typically web applications need an active connection to the Internet in order to work.

However data connection or Wi-Fi networks are often greatly disrupted (basements, subway) or simply not available (airplanes, remote areas). Even when they are available, there are factors like roaming charges, low bandwidth and battery consumption that also need to considered. Moreover, the user is free to disable the

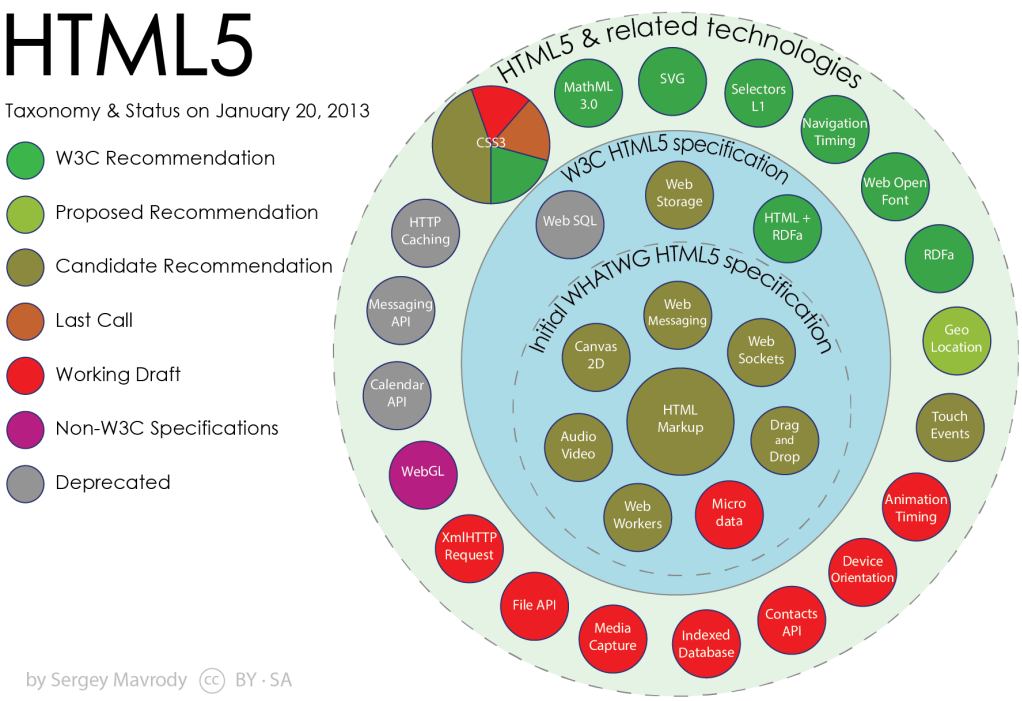


Figure 2.2: HTML5 APIs and related technologies. [10]

connection anytime. So designing an application that is usable and works reliably while *being offline* can be crucial.

Imagine the simple case where a person wants to log their meals to an application which then synchronizes the data with a remote server. Ideally one would like to perform the task even though they do not have a network connection at the moment, so that data is ready to be synchronized when a reliable connection becomes available.

A typical web application is stored in a remote server, so traditionally their pages cannot be accessed without an internet connection. The browser's cache can be used to store a small amount of data, but this is not enough for most cases.

The new HTML5 standard [9] specified two mechanisms that were to solve this problem: a Structured Query Language (SQL)-based database API for storing data locally, and an offline application HyperText Transfer Protocol (HTTP) cache for ensuring applications are available even when the user is not connected to their network [11].

As of the web SQL database, the W3C stopped working on them in November 2010 due to lack of independent implementations and have since been deprecated [12], although it is still supported in most major web browsers [13; 14].

As we have described in the previous section, native applications can be downloaded and saved on the device, simplifying offline functionality. Moreover, they

have access to the device's components, like camera, GPS or accelerometer, and they can also save large amounts of data to the file system. Hybrid applications can take advantage of this thanks to their native layer, which enables them to interact with the device just like a native application would do, including saving data into a local database.

Applications can take advantage of local storage in different ways. Some let their users *star* the content they want to have available offline, while others may do it automatically on the background, for a more seamless experience.

Working offline with an application can be an advantage most of the time, but it also has its drawbacks. For instance, since data is only synchronized whenever a network connection is available, it cannot be guaranteed that it is fully up to date at a certain point on time. Several clients can update the same data, so the user may not always have its latest version. Because of this, one also needs to consider the case where those clients update the same data simultaneously, a situation which can lead to conflicts and unreliable data.

3. NOSQL DATABASES

3.1 Why NoSQL databases?

Relational databases [15], unlike many other things, have been a constant in the computing world for decades and the number one choice for industry. Besides business and marketing reasons, there are a number of benefits that relational databases provide and have helped them stay at the top, such as persistent data, concurrency, integration and a mostly standard model [16, p. 3-5]. While many have tried to compete for their privileged position over the years, none have succeeded so far. NoSQL databases are a new challenger that is having a big impact in recent years.

The term NoSQL stands for 'Not Only SQL' and it was first used in 1998 by Carlo Strozzi, who gave this name to a new open-source Relational Database Management System (RDBMS) [17], but it was not until 2009 when Johan Oskarsson reintroduced the term in a meet up about "open source, distributed, non relational databases" [18; 19]. Since then, many NoSQL databases have emerged and their use by the industry has grown exponentially.

What is the main reason behind the rise in the use of NoSQL databases? In the latest years, the handling of so-called *big data* has urged companies to look for new ways of storing and manipulating that data, since the existing resources were not enough. NoSQL databases can handle large amounts of data since they are designed to provide a good performance while running on several clusters of small machines, something that relational databases are not designed for. This makes scaling up a system much easier and cheaper. In addition, big data is not only large, but also mostly unstructured. History data, metadata, log files, text, audio or video are some examples of unstructured data, which accounts for a big portion of the total data. NoSQL databases are able to cope with this data thanks to operating without a schema, which allows to redefine the database records freely, adding or removing fields without any changes to the structure.

This chapter will provide an insight of the main features, benefits and drawbacks of NoSQL databases by describing three of their main characteristics: their schemaless nature, distribution models and consistency.

3.2 Schemaless databases

The relational model requires data to be normalized beforehand, that is, a database schema is defined before storing any records to the database. This schema, as defined in [20], "is the structure described in a formal language supported by the database and provides a blueprint for the tables in a database and the relationships between tables of data". In other words, it is the logical definition of the database, a plan that defines the name of its tables and the name and type of data of each of the columns in every table.

On the other hand, NoSQL databases' approach is much more casual, allowing users to store any kind of data within the chosen model. This results in more freedom and flexibility, since the data can change over time according to the current needs, either by adding or removing things, operations that are much more laborious in relational databases, when a schema is binding the changes.

NoSQL databases can roughly be classified into four different categories, based mostly on the data model they use to store the information. These four different types are key-value pairs, column-oriented, document-oriented and graph databases. While most of them could solve the same problem, there will always be a *best fit* depending on the way data will be used, the storage space needs and available resources, among others. The following paragraphs describe the core ideas of each type.

3.2.1 Key-value databases

Key-value databases are the less complex and easiest to implement model of NoSQL databases. As the name suggests, data is stored so that a key is paired with a value. This structure can be compared to that of a map or hash table in a *regular* programming language. Because of this simplicity, key-value databases can perform extremely well in basic scenarios focused on get, put and delete operations based on a pair's key, but it is not so helpful when the queries and aggregations among data become complex.

While there are plenty of options to choose from, the most popular key-value databases, as published in the monthly ranking elaborated by DB-engines [21], are currently the free and open-source solutions Redis [22] and Memcached [23], the latter being used in several popular services such as Facebook or Reddit.

3.2.2 Document databases

Document databases contain *documents* which are made of a unique ID and values, very much like key-value pairs. However, values within a document may be of any kind, there are no limitations as long as they can be expressed as a document. This

allows complex structures such as nested documents within a single one and queries based on the values of the document rather than only their ID, greatly improving flexibility over key-value databases.

According to the ranking in DB-engines [24], the most popular document databases are MongoDB [25] and CouchDB [26]. The latter will be the focus of the case study in chapter 4.

3.2.3 Columnar databases

Columnar databases, also called column-oriented databases, are named after their most important design aspect. Traditional row-oriented databases, like any relational database, store the information related to a given row together. Instead, columnar databases store the data from a given column together, as shown in figure 3.1.

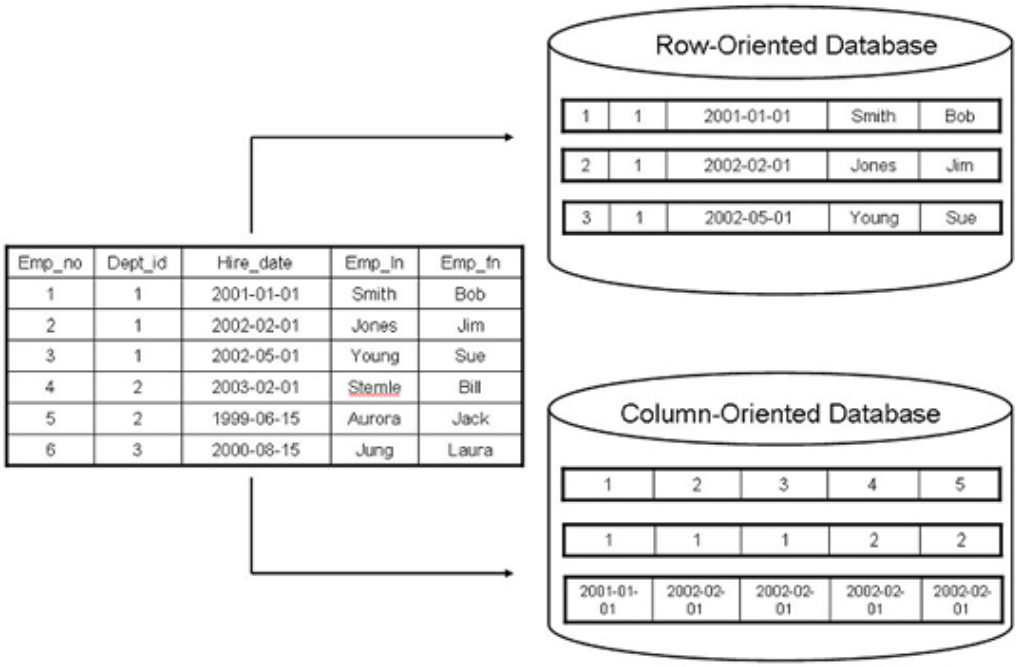


Figure 3.1: Comparison of data storage between column-oriented and row-oriented databases.

This change in the design has several implications, for instance, aggregations are much easier, since adding columns is inexpensive. Furthermore, it can be done only for those rows that need it, so each row may include a different subset of the columns or none at all. This also saves storage room as there are no *null* values, making columnar databases the most optimal solution for queries over large datasets.

Among the different columnar databases on the market, DB-engines cites [27] Cassandra [28] and HBase [29] as the most commonly used.

3.2.4 Graph databases

Graph databases are probably the less common of all the different types. As their name suggests, the structure of these databases follow that of a graph, with several nodes and relationships among them. Thus, graph databases are most adequate when dealing with interconnected data, which is stored as key-value pairs in both the nodes and the relationships. Figure 3.2 shows an example of a graph database with three nodes and six relationships.

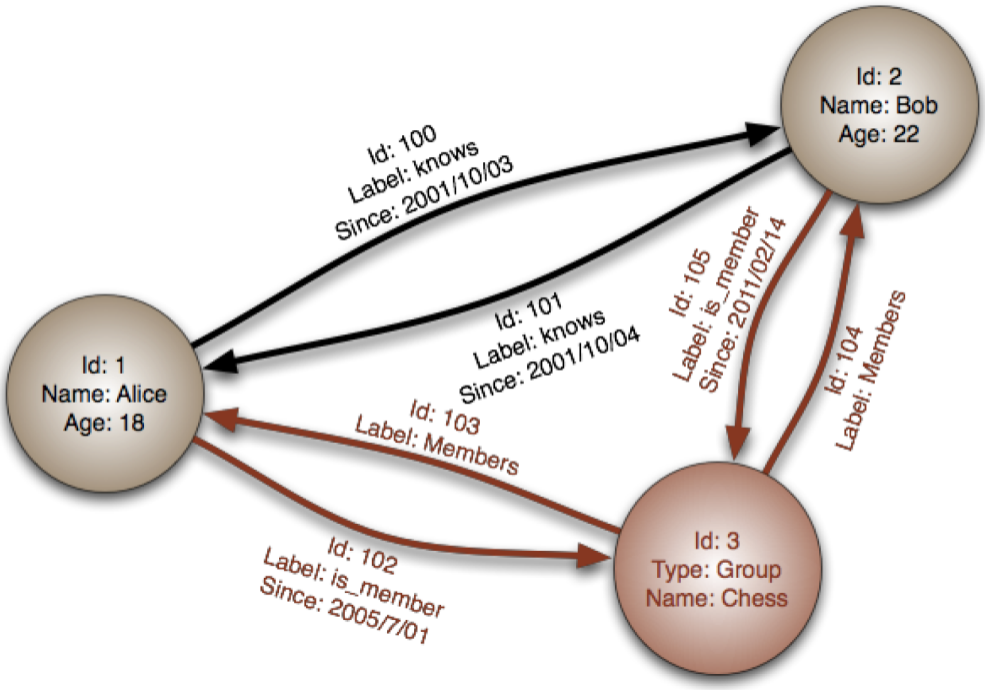


Figure 3.2: Example of graph database.

The most powerful argument of graph databases is their ability to transverse through the nodes by following the relationships, being able to efficiently solve any kind of graph algorithms, such as n-degree relationships or shortest path.

One differentiating aspect regarding graph databases is that they are better suited to run on a single server configuration, unlike the other data models, since almost every operation has to go through several nodes and relationships. We will address distribution models in deeper detail in the next section.

The most popular graph database at the time of writing is according to DB-engines' ranking [30] Neo4j [31], an open-source graph database fairly popular among

social networking applications, which are prone to present graph-alike relationships among their data.

3.3 Distribution models

As mentioned at the beginning of this section, one of the core principles of NoSQL databases is their ability to run on a cluster of servers instead of a single machine, unlike traditional relational databases, which are designed to be centralized in a single server. This helps scaling the system when the data becomes bigger and more resources are demanded. Just like the case of a data model, the use we are planning to make of our system will determine which model is better suited for our particular case.

As cited in [16, p. 37-45], there are two major techniques, sharding and replication, from which several distribution models derive. Essentially, sharding splits the information across different nodes, while replication duplicates the data over several nodes. The following paragraphs will cover four distribution models: single server, master-slave replication, sharding, and peer-to-peer replication. The following subsections describe them in further detail.

3.3.1 Single server

The most simple approach is having no distribution at all, that is, running the database and storing all data in a single server. This may well be the most adequate choice for a lot of applications which do not require any kind of distribution, and it avoids adding unnecessary complexity to the system since all the write and read requests are handled by the same entity.

3.3.2 Master-slave replication

Master-slave replication duplicates the information across several nodes, one of them being the master and the rest being slaves. The master node is the one responsible of updating the data, while reads can be done from either a master or a slave node. To ensure the slave nodes see the latest information available at any time, a replication process syncs the data from the master node to the slaves. Figure 3.3 illustrates the concept of this distribution model.

The master-slave replication model is specially beneficial when the information is subject to a majority of read operations. There are two main advantages in this case. First, it is very easy to scale the model by adding new slave nodes, which helps reducing their read load.

In addition, the slave nodes can still handle reads even if the master node fails, so even if writes cannot be processed, the scope of the failure is effectively reduced and

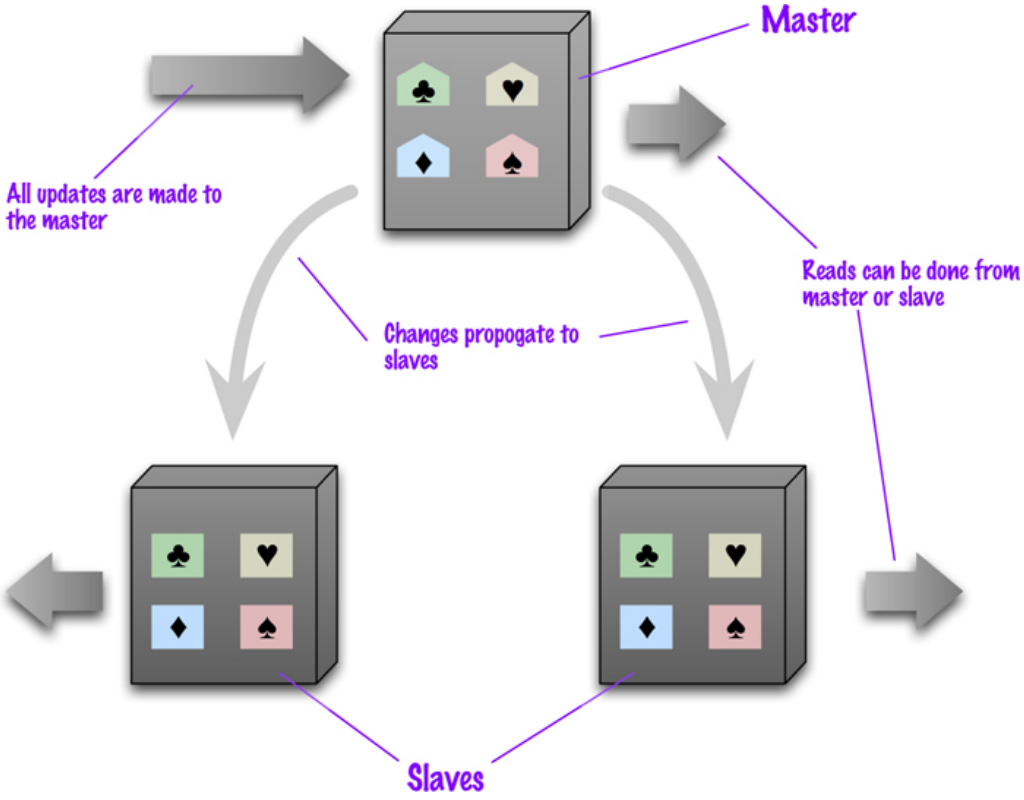


Figure 3.3: Example of master-slave replication configuration. [16, p. 41]

can be quickly recovered since any slave node can easily become the new master. This is known as *read resilience*.

On the other hand, there are some issues that make this model an inconvenient approach, specially if the system handles a lot of write requests. First of all, the master node can become the bottleneck of the model, since only it can handle all the write requests.

Moreover, in the event of failure in the master node, all writes will be lost. Once again, appointing a new master node would solve the problem, but some operations may have been lost already.

Finally, the synchronization process also has one major disadvantage, inconsistency. While this is an issue inherent to NoSQL databases, as we will see in the next section, in the case of master-slave replication different slave nodes can see different data at a given point in time if the master node has not yet completely propagated the changes.

3.3.3 Sharding

Sharding effectively splits the information into several parts and assigns to each of them a *shard*, a single server which will then process all the write and read requests associated to the data it contains, as can be seen in Figure 3.4.

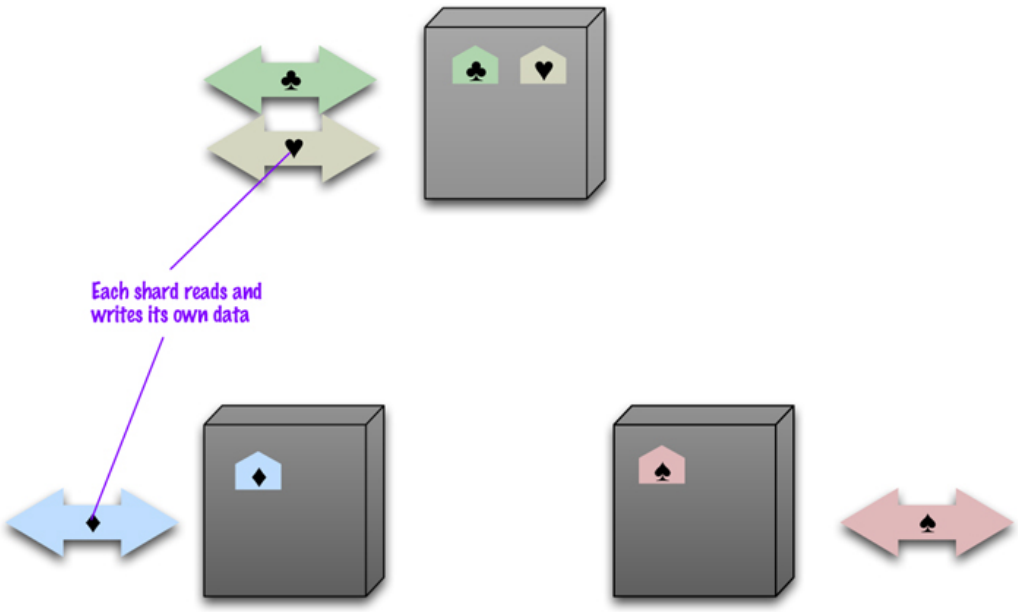


Figure 3.4: Example of sharding configuration. [16, p. 38]

Ideally, each shard would process approximately the same amount of requests so the load is well balanced among all servers. This is a difficult goal to achieve as the way data is accessed can depend on multiple factors, such as physical geographic location or simply the hour of the day. Some NoSQL databases feature automatic sharding, taking over the responsibility to spread the information across different servers automatically [32].

While replication served its purpose of improving read access to the data, sharding can greatly improve the performance as it takes care of both writes and reads, making it a better option if a system needs to handle a lot of write operations.

On the other hand, sharding is not much better than a single server system regarding failure tolerance. If one or several servers fail, the portion of the data stored in them becomes unavailable.

3.3.4 Peer-to-peer

Peer-to-peer is the most complex distribution model and is meant to solve one of the disadvantages of master-slave replication, that is, the ability to cope with systems with a heavy write load. Figure 3.5 shows how peer-to-peer works. The main idea

is to eliminate the role of the master node, essentially reading a number of replica servers able to process both write and read requests.

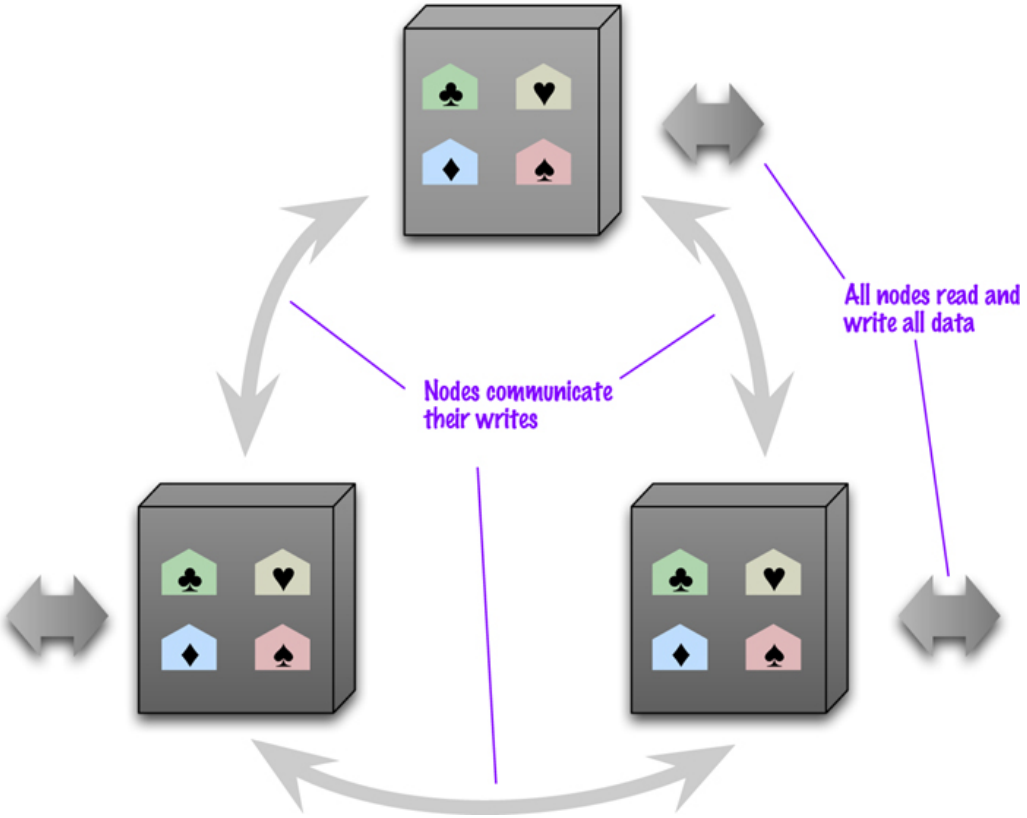


Figure 3.5: Example of peer-to-peer replication configuration. [16, p. 42]

At first sight, this model seems flawless. All the nodes have, theoretically, exactly the same data, so a failure in any of them does not prevent access to the data. It is also very easy to add new nodes to the model in order to improve performance.

However, peer-to-peer replication does not lack its complications. As with master-slave replication, inconsistency is a problem. The fact that every node is able to handle write requests means that eventually the same information might be updated at the same time in two different nodes, generating conflicts. Section 3.4 will further detail the inconsistency issues.

3.3.5 Combinations of sharding and replication

Not only it is possible to use one of the aforementioned strategies but they can also be combined for more complex models. Combining master-slave replication with sharding results in multiple master nodes, but with no information duplicated among them, since each information piece only has one single master.

In addition, combining peer-to-peer replication with sharding is also possible. Using this configuration data is shared among several nodes, but not all nodes have the same information.

3.4 Eventual consistency

In the previous pages we have seen some of the advantages that NoSQL, being designed to run on a cluster of servers, have. We have also talked briefly about one of the biggest disadvantages, consistency, which is greatly impacted by this design.

But, what is consistency in a database? Consistency is one of the four ACID properties and, as defined in [33], it "states that data cannot be written that would violate the database's own rules for valid data". In other words, every transaction should leave the database in a valid state meeting the database constraints, regardless whether they are defined by the user or by the database itself.

Traditional relational databases are able to offer strong consistency by avoiding the situations that may lead to inconsistencies in the database. Running in a single server also helps this purpose. Let's consider the simple case of the bank account balance:

1. Your bank account balance is 50€.
2. You deposit 20€.
3. Your new bank account balance is 70€ in *any* ATM.
4. You withdraw 10€.
5. Your new bank account balance is 60€ in *any* ATM.

This opposes to what occurs with NoSQL databases. Let's consider the following example to illustrate what eventual consistency means:

1. I tell you that tomorrow is Sunday.
2. Your neighbor tells his friend that tomorrow is Monday.
3. You tell your neighbor that tomorrow is Sunday.

We can see how, although *eventually* all the servers (represented in the example by you, your neighbor and me) know the truth about tomorrow being Sunday, a client (the neighbor's friend) has been left behind thinking tomorrow is Monday.

NoSQL databases need to relax consistency in order to be able to keep some of their other characteristics. The Consistency, Availability, Partition tolerance (CAP) theorem is the reason why.

The CAP theorem was first formulated by Eric Brewer in 2000 [34] and later proved in 2002 by Seth Gilbert and Nancy Lynch [35]. The theorem states that for any distributed systems, it is not possible to provide any more than two of the following guarantees at the same time:

1. Consistency, all nodes see the same data at any given time.
2. Availability, all nodes are able to read and write at any given time.
3. Partition tolerance, the system works even in the event of a network failure.

Figure 3.6 below illustrates the theorem, while showing where some of the databases we have talked about would be situated based on how they deal with each of the three properties.

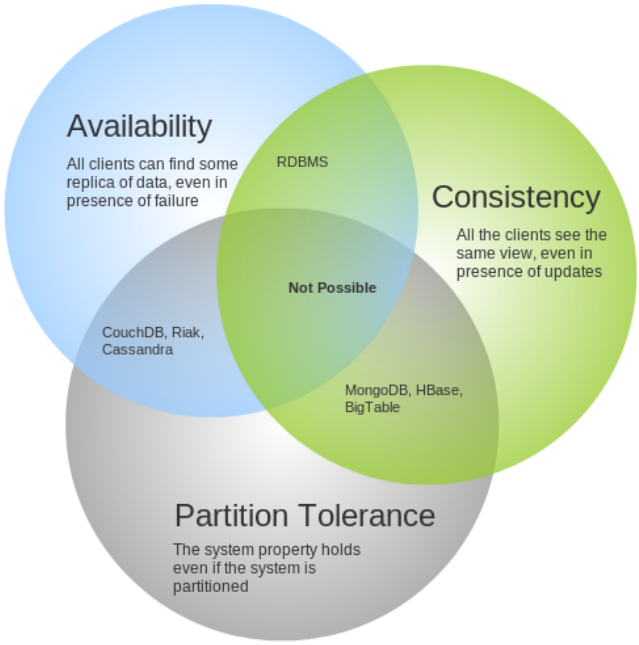


Figure 3.6: Illustration of the CAP theorem.

4. CASE STUDY: MOVENDOS PROTOTYPE APPLICATION

4.1 Overview

Before moving on to describe the details of the system, it is important to get to know it at a higher level and what triggered its development. This section describes the motivation behind the applcation and its overall architecture.

4.1.1 Motivation

Movendos Oy was established in late 2012 as the result of a research project that started at the beginning of that year at Tampere University of Technology's Biomedical Engineering department (later part of the Signal Processing department). The goal of the company is making personalized health coaching part of people's everyday life. Movendos' tool is aimed at personal health coaches who want to improve the effectiveness of their coaching, making it easier to assign tasks to their customers and be in constant communication with them.

Communication, as mentioned before, is one of the critical requirements of the application and is achieved through a system which allows coaches and customer to exchange messages, much similar to those in other services, such as Facebook.

Besides using this messaging system, coaches can assign different types of tasks to their customers, such as habit-forming tasks, where a user forms a habit by systematically repeating a task, monitoring one's weight, keeping track of sports or physical activities by logging them into the application or a simple diary task to write notes.

In addition, customers can also mark their results on the application's workbook upon completing a task, see a timeline of their recent and past activity, which also offers the chance to comment on it, and check their progress on a summary view.

The application shall be accessed like any other website using a web browser. This means that it should also be accessible using different kind of devices, such as desktop computers, tablets or smartphones, which also becomes a new requirement for the system. Moreover, there is a wish to develop a hybrid application for Android smartphones and tablets, which would take advantage of the native capabilities of those devices, such as notifications or possibility for offline functionality.

Before the development of the commercial product could start, there was a need to design and implement a prototype as a proof of concept. It also served to begin piloting with customers, whose feedback is valuable in order to iterate the design and requirements of the system.

4.1.2 Architecture

The system has two main components, application back-end and application front-end, which communicate with each other over a network connection.

Firstly, the application back-end is formed by the the Apache CouchDB server, which does the main work of accessing the database and supporting the front-end by providing the requested information. All information is stored in CouchDB databases, while the CouchDB REST API is the layer between those databases and the application front-end.

On the other hand, the application front-end is mainly composed by the user interface. It comprises all the HTML pages and Javascript files that interact with and are supported by the back-end. Within the front-end we can distinguish two distinct flavors of interacting with the back-end: the *regular* experience, using a Web browser, and the Android experience, using the hybrid application, which also provides extra functionality such as offline capabilities.

Figure 4.1 below provides a graphic description of the architecture of the system, with the different components identified. The following sections will provide an insight on each one of those components.

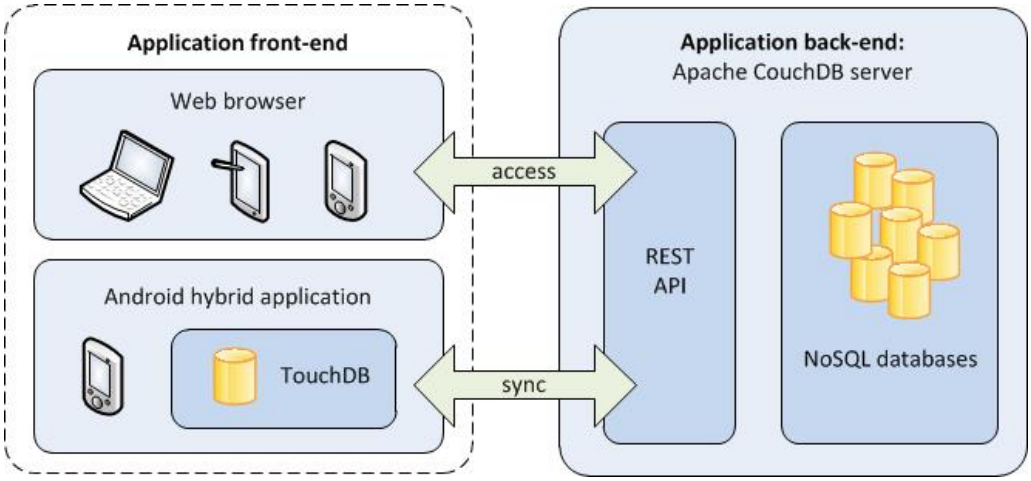


Figure 4.1: Architecture of the system.

4.2 Application back-end: CouchDB server

The application back-end is mostly formed by the CouchDB database. CouchDB [26] is a document-oriented NoSQL database based on JSON and REST, which was first released in 2005.

In section 3.4 we discussed how NoSQL databases in general, and CouchDB in particular, prioritize availability and partition tolerance over consistency, according to the CAP theorem. The first question that needed to be answered when considering CouchDB as the storage solution for the back-end was whether those characteristics suited the system. However, eventual consistency was good enough since the system does not have any kind of real-time requirements, so sacrificing consistency would not affect the user experience.

Movendos Oy decided to use IrisCouch [36] as the server host. IrisCouch offers an Apache CouchDB server with geospatial data capabilities. This server hosts all databases and their documents, providing a RESTful HTTP API in order to add, edit or delete them. We will describe the API in further detail in subsection 4.2.2.

4.2.1 Database types

We have seen how NoSQL databases in general and document-oriented databases in particular, allow users to store any kind of data into documents.

Documents in CouchDB are nothing but a JSON object with key-value pairs as fields. All documents have an `_id` field which uniquely identifies the document, and a `_rev` field that is updated every time the document changes. The `_id` can be set manually upon creation of the document or else CouchDB will generate one for us, while `_rev` is automatically managed by the database.

These two fields are extremely important as they are requested by the server whenever a document is to updated. If both the `_rev` and `_id` fields on the client side do not match the `_rev` and `_id` fields on the server side, the transaction is rejected. Because the `_rev` contains a sequence number which autoincrements with every update, CouchDB makes sure only the latest revision of the document is updated and avoids conflicts.

Besides those two special fields, the rest of the document can have any structure, including other fields, array structures, nested documents or even attachments.

Just like a regular e-mail, a document in CouchDB can hold one or several files as attachments. These can be either standalone or inline attachments. In the first case, a new document is created containing the raw data of the file. In the second case, the file is stored under the special field `_attachments`, as shown in Listing 1. The latter will be the alternative used in the application and has a major implication, as this is the way all the different front-end components are hosted.

```
{
   "_id":"attachment_doc",
   "_rev":1589456116,
   "_attachments":
   {
      "foo.txt":
      {
         "stub":true,
         "content_type":"text\/plain",
         "length":29
      }
   }
}
```

Listing 1: Example of an inline attachment.

One of the first tasks was to define a data model for the system. Even though CouchDB is flexible and changes can be relatively easily implemented later on, having a data model defined helps understand the problem and can save quite a lot of time in the long term.

Because of the different right access and nature of the documents, it was decided to split the documents into different databases, something that CouchDB makes it easy, either through its Web interface or command line tools. There is a total of four different types of databases in the system: common, personal, coach's and program storage databases. Each of these types is discussed in detail below.

Common database: The common database is public and contains most of the main application's data, such as all the HTML pages, images and JavaScript files that are used by the front-end, with the exception of task specific files. These files are saved as attachments to documents, so they are easily accessible, as we will show in the following section. The common database also contains a document which maps each user's username with their personal database name.

Personal databases: Personal databases store data which is particular to a certain customer of the system, and thus should not be accessible to everyone. Typically this data includes the personal profile, notifications, messages, the sets of tasks created specifically for the customer and their results. In order to easily identify these documents, they have predefined descriptive `_id` fields, such as the sample `profile` document shown in Listing 2.

It is also important to note that both messages *to* and *from* the customer are stored in the personal database. This has an impact while using the Android hybrid application . Only the customers themselves and their coaches are authorized to see the contents in a personal database.

Coach databases: Coach databases are those which store the data for those

```
{
   "_id": "profile",
   "_rev": "5-dc228fce6821ad04b6e457aeda0254ab",
   "type": "profile",
   "customerId": "lempinimi",
   "nickname": "Lempinimi",
   "company": "Movendos",
   "highLevelGoals": "Exercise more often",
   "created": "2012-09-18T12:05:03.000Z",
   "lastUpdate": "2012-11-06T07:49:14.244Z",
   "language": "fi",
   "role": "customer",
   "coachId": "coach",
   "coachNickname": "Ville Valmentaja",
   "lastSeen": "2013-08-22T07:49:14.244Z"
}
```


Listing 2: Example of a customer profile document.

users with a coach role in the system. Since messages that coach send to their customers are stored in their respective personal databases, the content of a coach's database is reduced to their profile, a list of the coach's customers, which is used in order to determine which personal databases a particular coach is authorized to read, and the coach's private notes about their customers, which are only visible to them.

Program storage: Program storage databases contain the templates used for each of the task types. These templates are HTML based so they can be embedded in the HTML pages stored in the common database, improving the flexibility of the system. Together with the templates there is usually a JavaScript file which handles all task specific functionality. We can see an example of a document in the program storage database in Listing 3 below.

As we can see from the examples, documents in the personal, coach and program storage databases have a common `type` field. This field was introduced to resemble some kind of schema feature in an otherwise schemaless database. Looking for document of a certain `type` defines which sort of fields and structure we are expecting to find.

4.2.2 Manipulating and querying the database

Once the data model has been defined, it is time to populate and interact with the database. As mentioned at the beginning of this section, CouchDB provides a RESTful HTTP API in order to add, edit or delete documents. This means all operations are done using HTTP requests, either through command line tools, namely the cURL command, or through the CouchDB Web interface, called Futon,

```
{
   "_id": "task_monitorWeight",
   "_rev": "6-0bccc539f7b24104aaefb29e0374967a",
   "taskHasCss": false,
   "taskFileName": "task_monitorWeight",
   "treatmentProgramName": "Painonseuranta",
   "type": "treatmentProgramTemplate",
   "treatmentProgramId": 3,
   "_attachments": {
      "task_monitorWeight.html": {
         "content_type": "text/html",
         "revpos": 6,
         "digest": "md5-LlgbhHbSyx/fElYVOJZHOA==",
         "length": 459,
         "stub": true
      },
      "task_monitorWeight.js": {
         "content_type": "application/x-javascript",
         "revpos": 2,
         "digest": "md5-nLEwODV8XuUAed5Rp5a5og==",
         "length": 1810,
         "stub": true
      }
   }
}
```


Listing 3: Example of a document in program storage.

which essentially performs the same operation. Normally, the URL to be requested looks like `http://server:port/database-name/_id`, but for illustrating the examples we will assume we are running the server on localhost port 5984, which is the default configuration.

Reading a document: In order to read a document from the database, we need to issue a `GET` request to the above URL, which results in the server returning the requested document:

```
$ curl http://localhost:5984/personal_db0000/profile
```

Creating a document: Creating a document is done by using `POST` on the database URL, without the `POST` of the document. We need to define the `Content-Type` header as `application/json` using the `-H` option, and include the document in the `-d` option.

```
$ curl -i -X POST "http://localhost:5984/personal_db0000"
-H "Content-Type: application/json"
-d "{ 'type': 'profile', 'customerId': ''lempinimi' }"
```

The server will response using code 201 if everything went successfully, and include important information such as the `_id` and `_rev` fields of the newly created

document.

Updating a document: Updating a document is done in a similar fashion as creating one, using `PUT` and specifying the `_id` of the document we want to update. It is worth noting that CouchDB overwrites the whole document rather than updating only those fields that are given in the request, so we need to copy all the information. This operation can also be used to create a new document whose `_id` we want to define manually.

```
$ curl -i -X PUT "http://localhost:5984/personal_db0000/profile"
-H "Content-Type: application/json"
-d "{ 'nickname': 'Lempinimi', 'company': 'TTY' }"
```

Deleting a document

Deleting a document in CouchDB flags it as deleted, rather than actually removing it from the database. Deletion can be performed using a `DELETE` request. Once again, the server will respond using code 200 if the operation was successful.

```
$ curl -i -X DELETE "http://localhost:5984/personal_db0000/profile"
```

Custom views: The previous operations showed how a single document is accessed, however, say we want to fetch all messages for a certain customer. How do we query multiple documents at the same time? Or how do we get all tasks whose end date is today? CouchDB offers the possibility to create custom views to query the database. Views consist of a map and a reduce functions that generate a sorted list of key-value pairs and are the main way to access documents in all but the trivial cases when a single document is required.

Creating custom views is done by defining a mapping function that CouchDB will run on every document in a database, passing each one of them as the `doc` parameter each time. This function must contain an `emit()` function, which will define the output. The `emit()` function takes two parameters: the first one will be the key, and the second one will be the value.

There is one predefined view in CouchDB, called `all_docs`, which returns all the documents within a database with their `_id` as key.

```
$ curl -i -X PUT "http://localhost:5984/personal_db0000/_all_docs"
```

```
{
   "total_rows": 1,
   "offset": 0,
   "rows": [{
      "id:": "profile",
      "key": "profile",
      "value": {
         "_rev": "5-dc228fce6821ad04b6e457aeda0254ab"
```

```
      }
   }]
}
```

In this case, the `key` and `id` values match, but this will not be the case very often. By default, CouchDB does not include all the data in the documents into the output, something that can be fixed by adding the `include_docs=true` parameter to the URL, except the information will be shown under a `doc` field rather than the usual `value`, which is now reserved for those fields specified in the `emit()`.

The equivalent function that achieves the same output as the `all_docs` view would look as follows:

```
function(doc) {
   emit(doc._id, {rev: doc._rev} );
}
```

Custom views can be saved in the database as design documents by specifying the design *document name* and the *view name*, since a single document can contain several views. Each design document has an `_id` that starts with `_design/`. By combining these elements a URL in the form of `http://server:port/database-name/_design/document-name/_view/view-name` is defined so that it can be queried to obtain the desired output.

Once demonstrated how the mapping function works, it is easy to create custom views for the different needs of the system. Let's discuss the case of the messaging view as an example. We would like to fetch all messages in a customer's personal databases, sorted so that newest messages come first. For this, we first check the `type` of the document, and emit only those that are of type `message`. As it was mentioned previously, the output is a list ordered by the key, in this case we want it to be `doc.time`, which is the creation time of the message. Declaring `doc` as the *value* to be returned effectively provides the same output than using the `include_docs=true`, so all information is available at the front-end. The resulting function looks like this:

```
function(doc) {
   if(doc.type == "message") {
      emit(doc.time, doc);
   }
}
```

This view is saved under `messages` design document, with the name `by_time`. These are combined and generate a URL that returns the following information when queried:

```
$ curl "http://localhost:5984/personal_db0000/_design/messages/_view/
by_time"
```

```
{
   "total_rows": 18,
   "offset": 0,
   "rows": [{
      "id": "b5f6503522ce74910716403331000b9e",
      "key":"2013-05-27T11:50:39.655Z",
      "value":{ [... document data ...] }
   }, {
      "id": "b5f6503522ce749107164033310022d3",
      "key":"2013-09-27T12:31:22.842Z",
      "value":{ [... document data ...] }
   }, {
      [... 15 other changes ...]
   }, {
      "id": "b5f6503522ce749107164033310022d3",
      "key":"2013-10-02T09:03:14.931Z",
      "value":{ [... document data ...] }
   }]
}
```

By default, CouchDB sorts the keys cronologically. We want the messages in reverse order, something we can easily fix by using the `descending=true` option.

```
$ curl "http://localhost:5984/personal_db0000/_design/messages/_view/
by_time?descending=true"
```

There are countless other options that provide a great degree of flexibility to the views. For instance, it is possible to limit the amount of messages the view returns using the option `limit=X`, where X is the maximum number of messages we want to obtain. This is specially useful when the amount is messages gets bigger and the queries take more time to return. At this point it is easier to fetch the messages in batches, something that can be achieved by combining `limit` with the `startkey` and `endkey` options. These options reduce the amount of results to those whose key are within the specified range.

For a complete list of views in the system and their respective mapping functions, please refer to Appendix A.

4.2.3 Creating notifications: changes API

Another important part of the system are the notifications. Just like in many other social sites, the application notifies the users when certain events happen. In our case, these events differ depending on the user role.

For customers, events that will be notified include new comments on the timeline, new tasks available, reminders for a task, missing the deadline for tasks and new messages on the inbox. On the other hand, coaches will get notifications about new messages and new comments on the timeline.

How does the system receives those events? CouchDB offers a changes API which watches a database for any change on its documents. Any time one or more documents are updated, the `last_seq` field of the database autoincrements. This field is used to keep track of changes in three different ways: polling, long-polling and continuous.

Polling: Simple polling of the Changes API with no parameters will return all the content of the database, just like using `all_docs`. The output, following the same fashion as views, does not include the document content unless we explicitly request it by adding the `include_docs=true` parameter to the URL.

```
$ curl "http://localhost:5984/personal_db0000/_changes"
```

```
{
   "results": [{
      "seq:": 1,
      "id": "profile",
      "changes": [{ "_rev": "1-dc228fce6821ad04b6e457aeda0254ab" }]
   }, {
      [... 63 other changes ...]
   }, {
      "seq:": 65,
      "id": "notifications",
      "changes": [{ "_rev": "23-nLEwODV8XuUAed5Rp5a5ogjm523noSf" }]
   }],
   "last_seq": 65
}
```

Typically we only want to get the changes that took place after the last time we checked. For this, it is possible to use the parameters `since` together with the `last_seq` index since which we want to start looking from. In order to get the new events, this value must be equal to `last_seq`. A lower number would cause duplicates in the notifications, since some changes would be noticed more than once, while a higher number would be expecting changes in the future.

```
$ curl "http://localhost:5984/personal_db0000/_changes?since=60"
```

Polling is the simplest approach where the application would check periodically for changes, which is ideal if we can accept a certain delay before notifying the customers.

Long-polling: For those cases where notifications should happen instantly, long-polling keeps the connection open for a certain period of time. Whenever an update happens, or after a certain period of time, the connection returns and is closed, so we need to issue a new request to start listening again. The response is the same as with the normal polling request. Long-polling is available by defining the `feed=longpoll` parameter in the URL.

```
$ curl "http://localhost:5984/personal_db0000/_changes?feed=longpoll
&since=60"
```

Continuous feed: Finally, the continuous feed can be used with the parameter `feed=continuous`. This feed returns all changes individually in real time and keeps the connection open until explicitly closed.

```
$ curl "http://localhost:5984/personal_db0000/_changes?feed=continuous
&since=60"
```

This option is good in order to reduce overhead. On the other hand, requesting the continuous feed asynchronously can make a web browser not receive the data until the connection is closed. In addition, unlike that of polling and long-polling, the response format is not plain JSON, requiring a bit more processing on the client side.

The alternative chosen for the application was long-polling. Whenever a user logs into the application, the latest sequence number `last_seq` checked is fetched from their profile and saved in a global variable. A single request for changes is then issued on the personal database, which is the only one relevant for the notifications. If some updates have happened since the last time the user was active, the request returns immediately with those changes. Otherwise, the request remains open waiting for changes.

In any case, eventually the request will return and the callback function is called. This function issues another request with the new `last_seq` number and passes the changes to the function responsible of handling them. Handling includes checking which `type` of document underwent changes, for which the `include_docs=true` is needed. Depending on the `type`, the corresponding notification is issued, if applicable. Below we can see a simplified version of the two main functions that deal with this process:

```
function bind_db_changes() {
   $.getJSON(PERSONAL_DB + "/_changes?since="+ lastSeq +"&feed=
   longpoll&include_docs=true",
      function(changes) {
         handleDBChanges(data, changes);
         bind_db_changes();
      }
   );
}
```

```
function handleDBChanges(data, changes) {
   for(i in changes.results) {
      if(changes.results[i].doc.type == 'message') {
         // Issue "new message" notification
      } else if(changes.results[i].doc.type == 'taskResult') {
         // Issue "new comment" notification
      } else if(changes.results[i].doc.type == 'task') {
         // Issue "new task available" notification
      }
   }
   lastSeq = changes.last_seq;
}
```

4.3 Application front-end

The application front-end is the part of the application the users will interact with directly, that is, the user interface. This section describes all the application views, how they were designed and the reasoning behind that design.

4.3.1 Application's views

One of the basic premises of the application was that it should be easy to learn and use. Navigation through the different views must be simple, and the important information should be accessible and visible at first glance. The target user range is wide and we can expect anything from young to elder people and experienced users to complete beginners with computers.

The application is composed of several views. Some of them are accessible to the coach only, while the customer views can be accessed by the customers themselves and their coach. The menu on the right hand side of the screen (in larger displays) provides easy navigation through all these views. Coaches and customer have a different menu, but coaches will additionally see the customer's menu when navigating through a specific customer's views.

Login page: Before the user can access any of the views, they should log into the system. This view presents a simple form to enter the username and password. Accounts were managed manually by Movendos and the users had no option to change their password in the application, neither be able to recover their password automatically in case of forgetting it. Logging our of the application happens by clicking on the link at the top right corner of the screen, which is always visible in every view.

Coach's main view (Valmennetavani): This is the landing page for the coach, upon logging in. The coach will see a list of their customers/coachees and links to four different views for each of them: timeline (aikajana), messages (viestit), training program (ohjelma) and profile (profiili), as seen in Figure 4.2. Customers

can be sorted by different criteria such as name and company name, and text filter is available to search for a certain customer. The coach can also see and edit their private notes through this view.

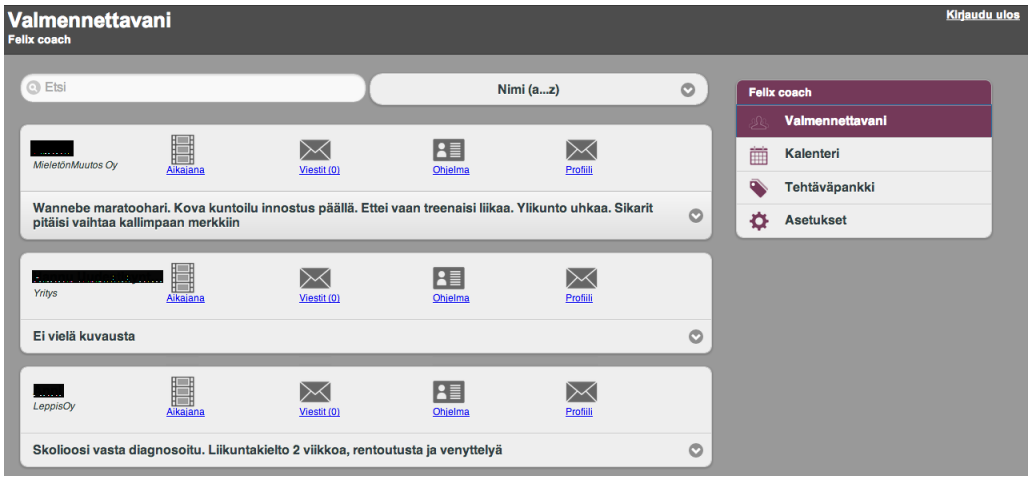


Figure 4.2: Coach's main view.

While the coach's menu shows links to calendar (Kalenteri), task library (Tehtäväpankki) and settings (Asetukset), those are disabled as those features were not implemented in the prototype. In addition, the link to the profile view of a customer was inactive, since that information was included in the customer version of the training program view.

Training program (Ohjelma): The coach's version of the training program view is accessible trough the corresponding link in the main view, Here the coach can see a list of active, future and past tasks for the selected user, edit the existing tasks and assign new ones using the task editor.

Task editor (Tehtävän muokkaus): Upon selecting a new or existing task in the training program view, the task editor is shown. This view allows the coach to edit the necessary details of the task. Although the details vary, tasks usually include at least a description that can be edited to help the customer understand the purpose of the task. Also common are the start and ed dates of the task and the scheduling and reminder options.

A task can be scheduled to be done whenever the customer wants, a fixed number of times, every day, certain number of times a week (with the option of specifying also which precise weekdays) or on precise given dates. Whenever the given time frame is over, a task is considered as missed if the customer did not record a result for it.

As for the reminders, they can be set to any day of the week at any time. Multiple reminders can be set for the same task, and a reminder message can be specified so

the corresponding notification has relevant information. Reminders can as well be completely disabled.

Workbook (Työkirja): The workbook is the landing page for customers. Here they can see a list of active tasks, fill in their results and save them to the timeline. Each task is presented as an expandable bar which opens up upon clicking. Every task type has a different icon that makes it easy to identify them. Once a task is expanded, the actual workbook is expanded. Depending on the task type, different input fields are shown. Saving the results stores the data in the database, which will then be accessible through the timeline and summary views. Figure 4.3 shows an example of workbook view.

Figure 4.3: Workbook view with sample tasks.

Summary (Yhteenvedot): The summary page contains a summary of the tasks so customer can easily check their progress. Each task type present the information in its own way. It is possible to take a snapshot of a summary and save it to the timeline for further commenting, as comments are not allowed in this view. It is an important view specially for the coach's evaluation process, and its main purpose is to clearly display the progress of the customer at a simple glance. Figure 4.4 shows how this view typically looks like.

Timeline (Aikajana): The timeline acts as a log of the customer's activity and interaction with the application. Every task result is displayed in this view in reverse chronological order, that is, newest first, as can be seen in Figure 4.5. Missed tasks (those which have not been completed before their end time) also create a new item on the timeline.

To improve clarity, only the ten most recent activities are displayed, while the rest are grouped by month in expandable lists. Each item in the timeline has a comment field where both the customer and coach have the option to discuss about

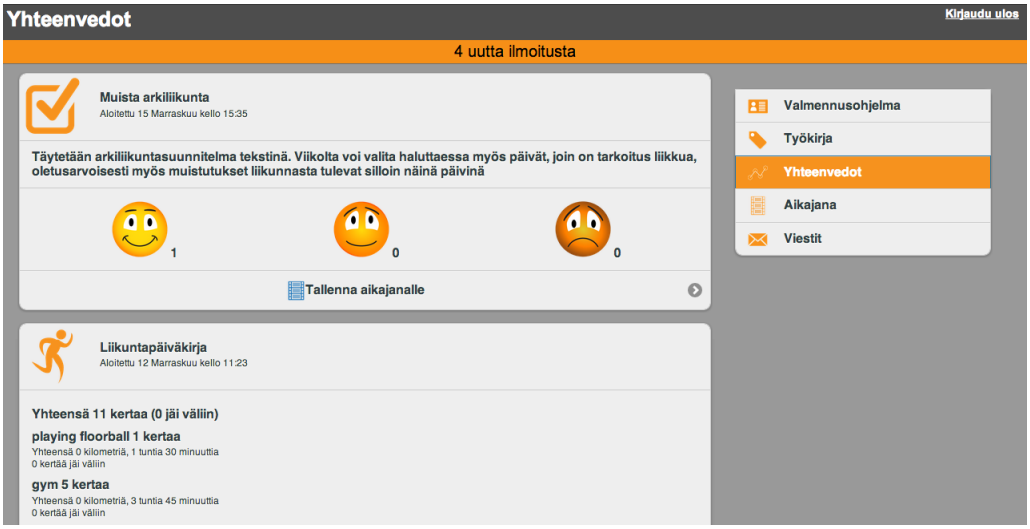


Figure 4.4: Summary view.

that item. Notifications are issued whenever a new comment is entered.

Filtering of the items on the timeline is also available through an input field at the top of the view, so it is easier to find a certain item by looking for the correct task name.

Messages (Viestit): From the messages view both customer and coaches can send messages to each other. Communication and the possibility to keep in touch easily was one of the core features of the application. The view presents the more recent messages in chronological order. A limit was set in order to improve the response time of the page by avoiding loading all messages at the same time. However, previous items can be loaded on demand using the button at the top of the view. Figure 4.6 illustrates how this view looks like for both customer and coach.

Training program (Valmennusohjelma): Customers can see a version of the training program view which differs from the coach's one. In this view the customer can set their personal goals and profile picture, as well as seeing a list of active and future tasks and a progress section, which shows how many tasks have been successfully completed over the last two weeks. Figure 4.7 shows an example of this view.

As explained previously in subsection 4.2.3, notifications (ilmoitukset) are another important part of the application. Once any number of notifications has been issued, they are readily available from any view in the application by clicking on the orange notification bar at the top of the screen. This displays a popup with a list of notifications. Each notification is itself a link to the relevant view they address. For instance, a notification of a new message will link to the messages view, while a reminder of a task will link to the workbook. We can see an example of notifications

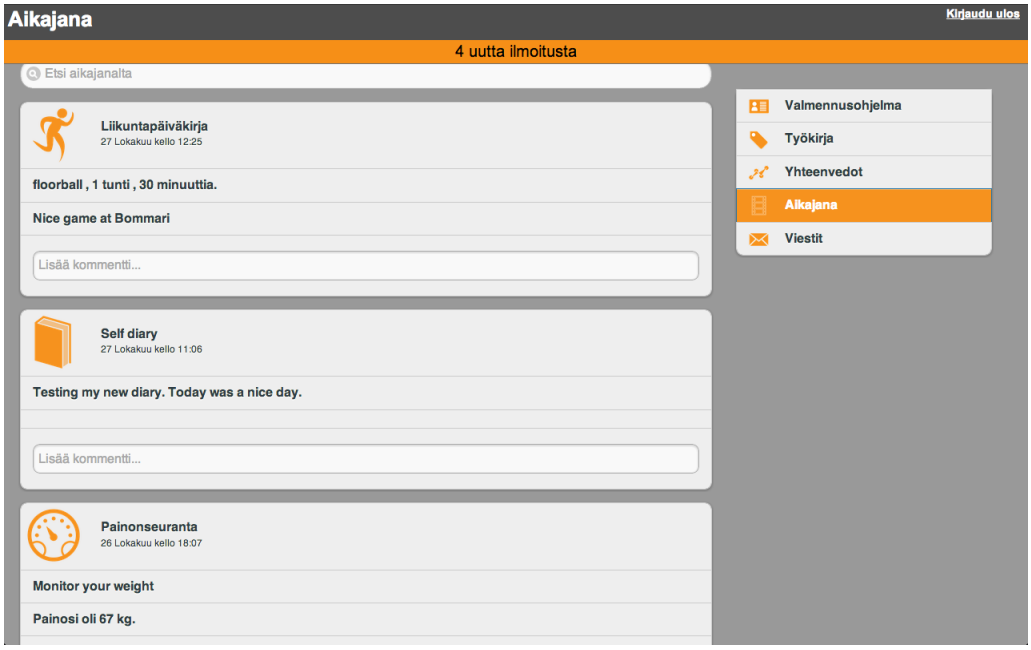


Figure 4.5: Timeline view with recent activity.

in the system in Figure 4.8 below.

4.3.2 Responsive UI: jQuery mobile

One of the main requirements of the application was to be able to run on a wide range of devices, from desktop computers to smartphones, including tablets, notebooks and basically any devices able to access the Web. Needless to say, every device type has a very different screen resolution, being able to display more or less content. Moreover, the interface can also be different. While desktop computers typically use a physical keyboard and mouse to interact with the content, smartphones and tablets usually have a touchscreen, which can be operated using one's fingers or with a stylus pen.

It was then critical to find a solution that works across all these devices, and it was decided that jQuery Mobile [37] would be used for the project.

jQuery Mobile offers a HTML5 based user interface built on top of jQuery and jQuery UI. It works on all major platforms, including all the more popular versions of Apple's iOS, Android or Windows Phone, as well as desktop platforms and Web browsers.

It offers a big selection of ready-made components that are used using simple HTML tags and properties. Among these components there are lists, buttons, form elements, icons or dialogs, to name a few. It also provides a library of customer events, that can be captured by the application to react to certain user actions.

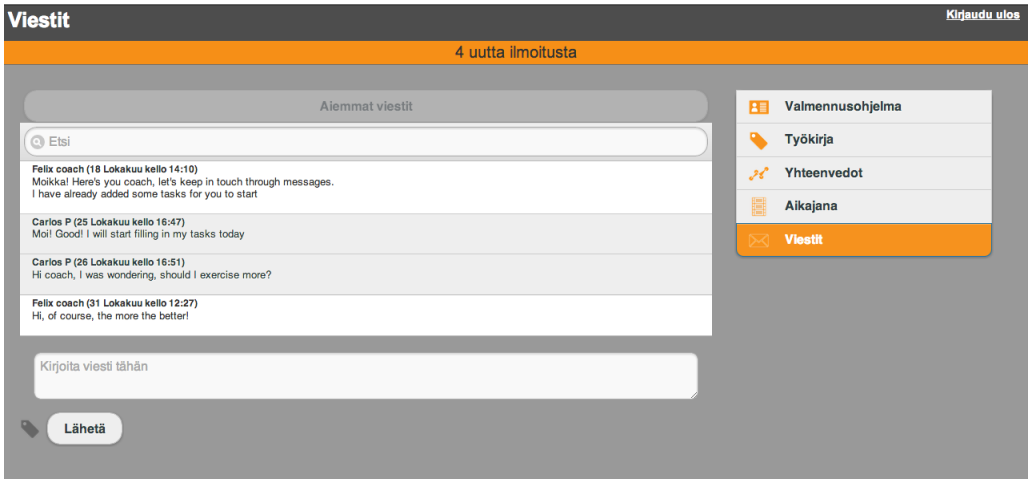


Figure 4.6: Messages view.

These events are also focused on mobile devices, and as such include actions like touching, swiping the screen or changing the orientation of the device.

In addition, it features a theme creation tool named ThemeRoller [38] which makes it easy to create custom themes by automatically generating the CSS style definitions.

Not only finding a suitable framework was important for the application, but also how to arrange those components in the screen based on its size. Depending on the reported display wide resolution, there are three possible layouts for the application. Each layout arranges the same elements in a different way, as can be seen in Figure 4.9.

For displays more than 1300 pixels wide, usually those in desktop or bigger laptop computers, the layout is divided in three columns. The menu is displayed on the right part of the screen, while the middle and wider column holds the main content. To avoid this column to be too wide, there is extra empty space at the left part of the screen.

For smaller displays between 600 and 1300 pixels wide, the empty space at the left of the screen disappears, leaving only two columns so the main content is aligned at the left of the screen. The rest of the elements remain at the same position. This layout is meant for smaller laptop displays and tablets in landscape mode.

Finally, for those displays less than 600 pixels wide, found in smartphones and tablets in portrait mode, the layout changes completely. There is only one column taking the whole width of the screen, showing the main content of each view. The menu completely disappears from the screen and is hidden behind a button at the top left corner of the screen. Pressing this menu button makes the menu slide from the left side of the screen, sweeping along the main content column. Pressing any of

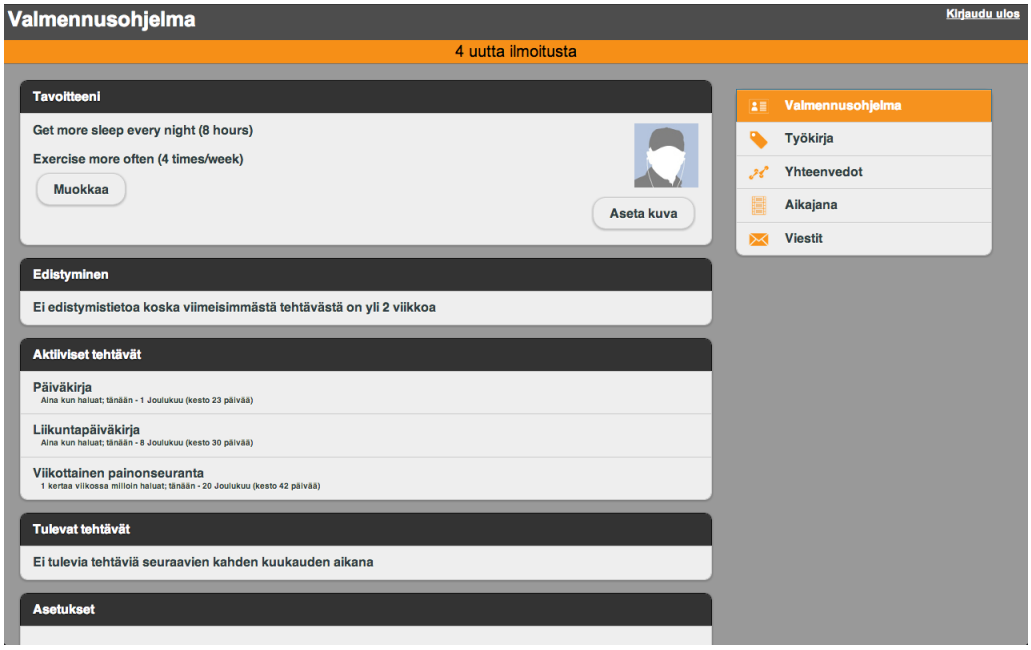


Figure 4.7: Training program view.

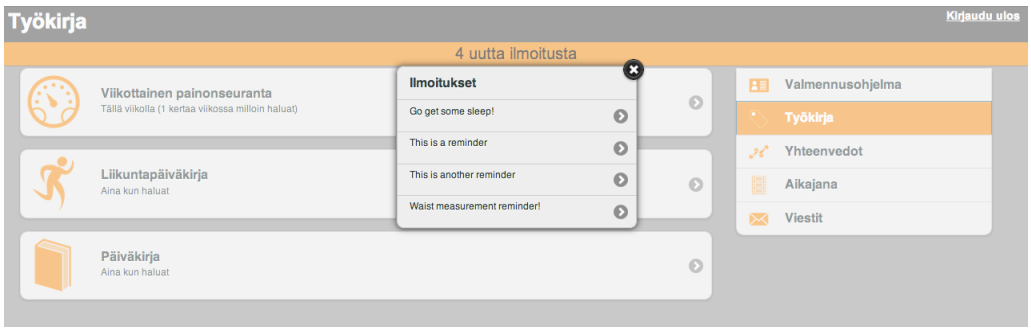


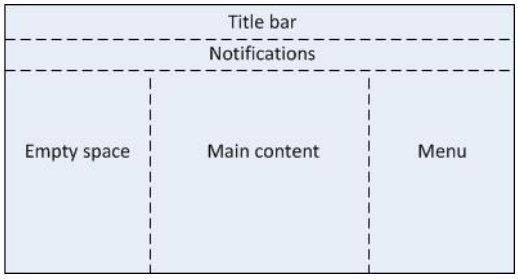
Figure 4.8: Notifications popup.

the items in the menu or the menu button itself restores the initial layout. Figure 4.10 below shows both desktop and mobile menus.

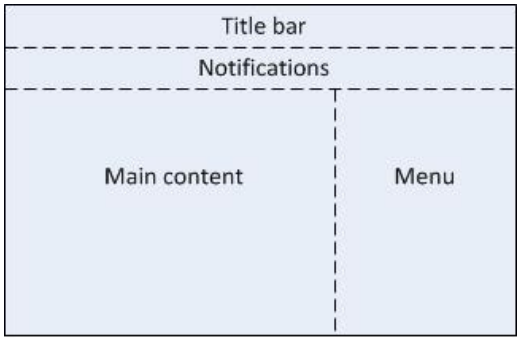
4.4 Android hybrid application

The use of smartphone applications has become a daily habit for their users, making it possible to achieve many tasks on the device. By developing an exclusive application for Android smartphones, we tried to make it easier and encourage customers to use the system.

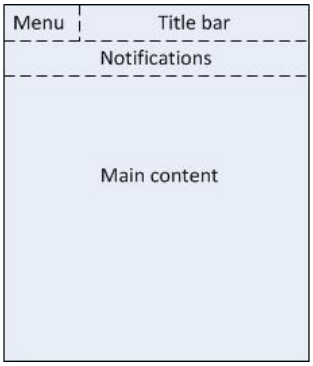
How to develop an Android application is out of the scope of this thesis and this section will focus in the two main challenges we tried to complete: accessing the built-in device components through the native API and providing offline function-



(a) Desktop/bigger display.



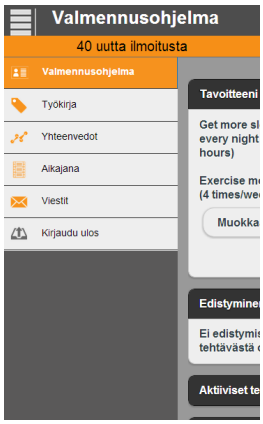
(b) Tablet/smaller display.



(c) Smartphone displays.

Figure 4.9: Comparison of the application's layout for different display wide resolutions.

(a) Desktop menu.



(b) Mobile menu.

Figure 4.10: Comparison of application's main menu on desktop and mobile devices.

ality to the users, as well as providing an overview of the application.

4.4.1 Overview

The approach taken into the Android application was to design a hybrid application that would offer optimised mobile performance and functionality. Choosing a hybrid application instead of a native one would allow us to reuse the code already written for the regular Web application, leaving only the native layer to be implemented.

Google's operating system Android, according to recent studies, accounts for as much as 81% of the market share of smartphones [39], so it was the preferred platform for development, instead of other popular choices like iOS or Windows Phone.

Supporting a certain level of offline functionality was another reason to develop the application, something that could be achieved by storing the information locally in the device.

It is important to note that the Android application was meant to be used by customers only and not coaches. The reasoning behind this decision was that customers could easily fill in their task results on their phones wherever they were, as well as receive native notifications on the phone. On the other hand, coaches would be likely to find working on a smartphone display quite uncomfortable, since the usage they make of the application is completely different.

4.4.2 Java interface: Accessing the native android API

As we described in section 2.1.3, hybrid applications are able to access the device's native functionality and built-in components through by encapsulating a mobile optimized web application into a native package. In our particular case, the native layer will allow us to access the camera and native notification systems.

Camera: The camera functionality was required for the "Food diary" task type, where the customers are able to upload a picture of their meals to the server as an attachment to their task results. Accessing the camera from an Android application is a simple task thanks to the existing API. The first thing to do is to add a camera permission in the manifest file. The `android:required="false"` option is added because the application can work also when a camera is not available.

```
<manifest>
   <uses-feature android:name="android.hardware.camera"
android:required="false"/>
</manifest>
```

The next step is to implement the function that actually does the work. Just like with many other native capabilities, Android uses an `Intent` for delegating the action on the camera application. This `Intent` is passed as a parameter to the `startActivityForResult` function.

```
public boolean takePicture(String whereToSave) {
    File file = new File(whereToSave);
    Uri fileUri = Uri.fromFile(file);
    Intent cameraIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    cameraIntent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
    startActivityForResult(cameraIntent, CAMERA_REQUEST);
    saveLocation = fileUri;
    return true;
}
```

The above functionality is enough so that the application is able to capture an image using the camera application, but that is not end of the job, as we still need to handle the results. As we can see, the `startActivityForResult` takes two parameters. The second parameter is used by the `onActivityResult` function, which receives the results from the camera application and takes the required actions. In this particular case of the camera, this involves checking the size and dimensions of the image, scale it if needed and saving it to the file system for future reference.

Native notifications: Notifications in Android are messages displayed in the device's notification area as an icon and in the notification drawer as a detailed message, instead of the actual application UI. By using notifications, the users can get important information from an application without having to open it. We can see an example on how the notifications look like in Figure 4.11.

(a) Notification area. (b) Notification drawer.

Figure 4.11: Example of notifications on Android devices [40].

Notifications in the system are created in the `MovendosNotificationManager` class. The looks and text of the notification are defined using an object of class `NotificationCompat.Builder`, which has methods that allow defining all kinds of options regarding look and behavior, some of which are mandatory.

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this.parent)
    .setSmallIcon(R.drawable.ic_launcher)
    .setContentTitle(title)
    .setContentText(text)
    .setNumber(number)
    .setTicker("Uusia ilmoituksia")
    .setWhen(time)
    .setLights(Color.CYAN, 1000, 1000)
    .setPriority(2)
    .setContentIntent(contentIntent)
    .setAutoCancel(true);
```

Issuing the notification is achieved by passing the created object the `notify` function in the `NotificationManager` class.

```
mNotificationManager.notify(mId, mBuilder.build());
```

In our particular case, notifications are handled not by the main `Activity`, but rather by the background `Service` constantly running on the device. If the `Activity` was the responsible of handling the notifications, these would become unavailable as soon as the application was closed by the user. Hence, we needed a `Service` instance to ensure there is a process continuously running in the background checking for new events. This service performs the same tasks as its equivalent on the CouchDB server, the `bind_db_changes` function described in section 4.2.3.

4.4.3 Offline functionality and replication

One of the main benefits of using the Android native package over the Web version application is the possibility to work without an active network connection. In order to offer this feature, there was a need for storing resources locally in the device, so they would be accessible without the need to request them from the database. This was achieved by replicating the data from the main server into the device.

TouchDB is a "lightweight Apache CouchDB-compatible database engine suitable for embedding into mobile or desktop apps" [41]. As stated in that same source, "if CouchDB is MySQL, then TouchDB is SQLite." It is able to replicate with CouchDB and has almost the same REST API as this one has, although not every feature is supported.

The distribution model between TouchDB and CouchDB resembles a master-slave replication model, although it is not a typical one. If we think of it that way, the CouchDB server acts as a master node, while the TouchDB would be a slave. However, writes are done locally on TouchDB before they are sent to the master

Of all the databases on the main CouchDB server, only two get replicated to the device. The first one is the common database, which includes all the web views,

JavaScript files, images and other UI elements. The second one is the customer's personal database, which contains all the data which is specific to that customer, including their tasks and profile information.

Replication works differently for common and personal databases. The common database gets no writes from the user and only gets updated either when there is a "software update" (for instance a new task type is added or a view gets a new feature), or when new customers are added to the system so the mapping document is updated. This means that replication works in one way only, from the main CouchDB server to the device.

On the other hand, the personal database is constantly subject to writes from the user, which can range from updating their profile picture to filling in a new task's results. Those changes, when made directly on the Android application, are first written locally to TouchDB, and then sent to the main server. In a similar way, changes made from the Web application or another Android clients are also replicated from the main server to the device, resulting in a two way replication between both servers. A diagram on how replication works on the application can be seen in Figure 4.12.

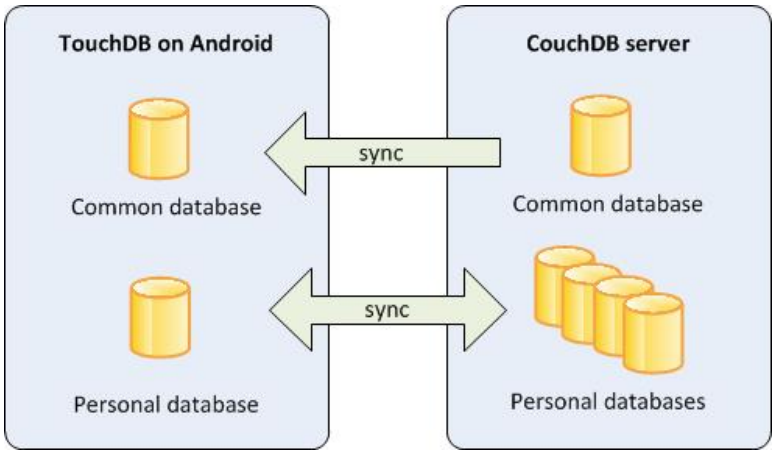


Figure 4.12: Replication between CouchDB and TouchDB.

The whole replication process can be divided in two. After the user has installed the application and boots it up for the first time, the first two steps are logging in using the credentials assigned to them and saving the username and password to the storage preferences. After that, the initial replication process between the main CouchDB server and the local TouchDB database starts, and the application follows the following steps:

1. Request the mapping document on the common database from the CouchDB server to find out the name of the customer's personal database.

2. Initiate one time pull replication for the personal database.
3. Initiate one time pull replication for common database.
4. Store "initial replication done" to local storage.

Once the initial replication has completed, the continuous replication process starts, which will update the information whenever changes have been made and a network connection is available. As stated before, the replication for the common database is only one-way, while the personal database is two-way. There are two steps the application performs in this process:

1. Initiate the continuous pull replication for common database.
2. Initiate the continuous push and pull replication for personal database.

This whole process can be seen in the sequence diagram displayed in Figure 4.13 below. 'Core SW' identifies the application's main activity. The web views the user sees on the screen are, in chronological order, 'init.html', 'login.html', 'please-wait.html' and finally 'workbook.html'. The HTTP listener is responsible for initiating the TouchDB replication, which synchronizes the data with the CouchDB server.

One important issue to consider and which caused problems with the certain old devices was the amount of memory they have. During replication, relatively big amount of data are transferred and this can cause certain devices to run out of the memory, which in turn makes the application crash. In order to solve this problem, replication was tweaked so that not every single document within the databases was replicated, but only those which were predicted to be used in the near future. for instance, fetching messages that were sent months ago is not required, so a date limit is set and anything that falls behind that limit is fetched upon request, assuming there's a network connection available.

Another challenge faced when dealing with replication was the different sequence numbers in TouchDB and CouchDB. We addressed in sections 4.2.1 and 4.2.3 about the `last_seq` field in a database, a sequence number which autoincrements with every update made to it. This number was mainly used to check for notifications, and for this to work correctly it is critical that the number is exactly the same as in the database.

During the replication process, the sequence numbers in TouchDB and CouchDb got out of sync. likely due to some operations not been replicated to TouchDB. For instance, if a document is created and soon deleted in the main server, the CouchDB database's sequence number will reflect those two operations, but there is no need to replicate those on TouchDB.

When fetching notifications, a request to the changes API was issued using the `since` URL parameter and the value stored in the customer's profile. While the

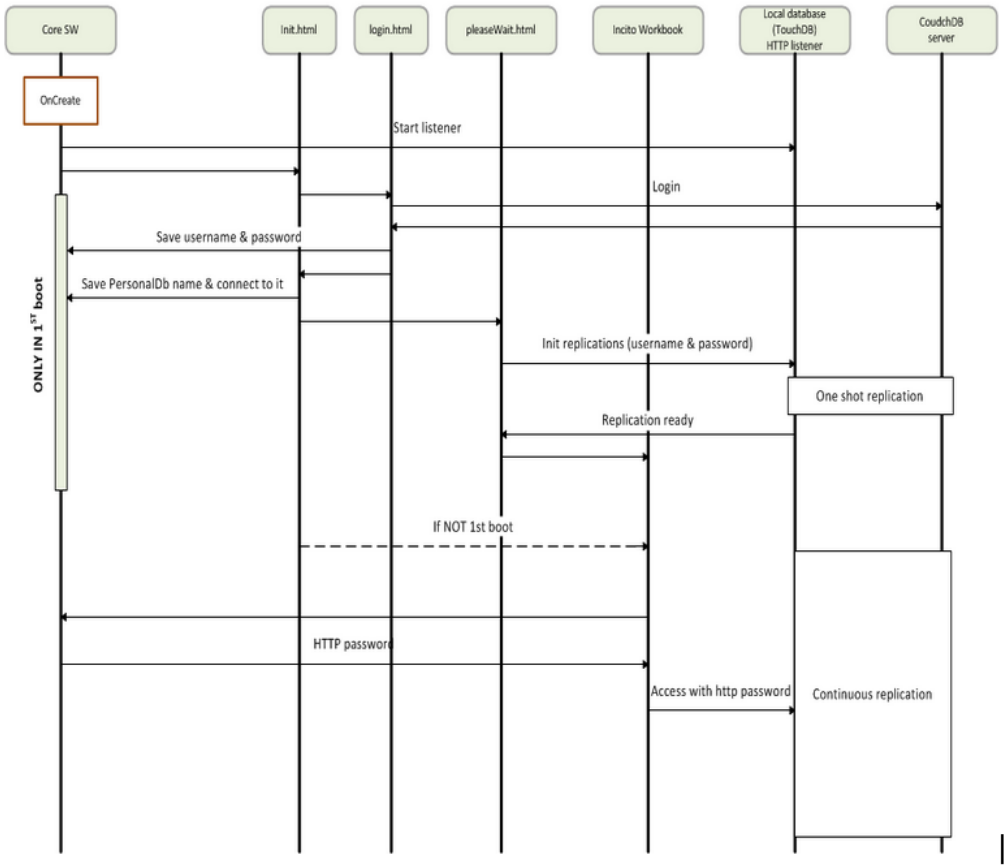


Figure 4.13: Login sequence diagram for Android application.

fact that both `last_seq` fields on TouchDB and CouchDB databases were not equal did not affect the regular Web, it was causing major problems on the Android application, as the the value stored on the customer's profile was usually higher than the actual `last_seq` field value on the TouchDB database, which meant the application got no notifications at all.

In order to solve this problem, a new `last_seq_touchDB` field was added to the customer's profile, which keeps track of the `last_seq` value exclusively for the TouchDB database.

5. EVALUATION

This thesis presented to the reader the concepts of NoSQL databases, cross-platform development and offline functionality in mobile applications and how they can be put together to develop a functional prototype application for an online health coaching tool that will be used by professional health, wellness and rehabilitation coaches. This application was meant to be used either through Web browsers on any kind of device or using an Android application for an optimized mobile experience, in those devices with Android installed.

The highest level results are those that the pilot conducted by Movendos Oy during the spring of 2013 using the developed system yielded. It gave the company lots of valuable feedback to be considered when developing the commercial software. Most of this feedback was related with front-end issues and general performance, but also some parts of the concept itself were debated. The Android application was initially not available for customers as its development took longer than planned, but after a few weeks some of them started using it.

Users	23 (of which 2 coaches)
Tasks created	105 (avg. 5)
Tasks completed	573 (avg. 27,3)
Tasks missed	297 (avg. 14,1)
Comments written	172 (of which 18 by coaches, avg. 7,5)
Messages sent	120 (of which 63 by coaches, avg. 5,2)

All in all, the results of the pilot and the feedback from the users were very positive and that has allowed and encouraged Movendos Oy to successfully develop the first commercial version of the product which, as of November 2013, has already been released.

As for the back-end, choosing a NoSQL database as the storage solution for the application has proved to be the right one. They suit perfectly for an experimenting system like this prototype, mainly because of their flexibility. Using CouchDB in particular was a good option for this application, based mainly on how it handles availability, partition tolerance and consistency. However, the most important point that can be extracted is that NoSQL databases have different characteristics and offer a very diverse range of features among them. The question of which one to use depends on the specific needs, and sometimes one can even consider combine them

to get the most out of them.

Regarding the front-end, most views were valuable and have been kept as they are, but other have been reconsidered. Looking at the statistics of usage of the prototype it was clear that messages were more valuable and used by the coaches, while users tend to use more the comments on timeline. This can be easily explained because of the fact that coaches do not have the time to check all of their customers' timelines and each of their items. Coaches also complained about not being able to easily follow a customer's progress. This ultimately resulted in the removal of the timeline view as a whole, giving more importance to the summary view.

Also targeted was the scheduling of the tasks and the impossibility to edit old task results. Some customers criticized that sometimes they were not able to fill in their task results before the task's end time, even if they had actually done them on time. This resulted in a lot of missed tasks in the system, that could not be edited afterwards. Mistaking some information when filling in a task result could not be fixed either once it was saved. These two suggestions also had an impact on the commercial released, which allows adding and editing task results in the past.

The Android hybrid application was likely the most controversial part of the system. It was developed as an extra feature in order to provide an optimized mobile experience. Android devices are indeed the most widespread nowadays, so platform wise it was a good decision to focus on it. The decision of using a hybrid application also allowed to reuse as much code as possible from the web application.

The possibility to use the application while being offline was achieved by using Touch DB on the device so operations between the front and back-end would remain local, while a synchronization process running on the background makes sure data is consistent with that on the CouchDB main server. This allowed faster performance when compared to operations over mobile network. However, there were plenty of other local performance issues, such as the initial replication problem, re-indexing and low RAM memory on older devices that ended up spoiling the user experience.

Another point to consider is that TouchDB was a technology defined as a *'labs' project* by their own creators, and while it has reached a stable release, it was probably not mature enough at the time of developing, which may explain some of the issues that were encountered. After that stable release, the project has been upgraded and renamed as Couchbase Lite [42], continuing the path that TouchDB started but with several architectural changes and providing a bigger set of features.

In our experience, hybrid applications combine the best of both native and web experiences, but also the worst. The fact that the user has to be bothered to install an application which then lacks performance over its web counterpart makes us debate whether it was the ideal solution for this particular application.

BIBLIOGRAPHY

[1] Tony Wasserman. "Software Engineering Issues for Mobile Application Development" [online]. 2010, FoSER [accessed on 19.02.2013]. Available at: http://works.bepress.com/tony_wasserman/4

[2] Qt digia. [online] [accessed on 12.11.2013]. Available at: http://qt.digia.com/

[3] Apache Cordova. [online] [accessed on 12.11.2013]. Available at: http://cordova.apache.org/

[4] W3C - HTML. [online] [accessed on 12.11.2013]. Available at: http://www.w3.org/html/

[5] W3C - Cascading Style Sheets home page. [online] [accessed on 12.11.2013]. Available at: http://www.w3.org/Style/CSS/

[6] Mozilla Developer Network - JavaScript. [online] [accessed on 12.11.2013]. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[7] Shahar Kaminitz. "Native, web or hybrid mobile app development?" [online]. 2011 [accessed on 23.02.2013]. Available at: http://www.scribd.com/doc/50805466/Native-Web-or-Hybrid-Mobile-App-Development

[8] Sebastien de Mel. "AppTalk Frontline: Web vs Hybrid vs Native" [online]. 2013 [accessed on 19.04.2013]. Available at: http://www.slideshare.net/WallRushGO/apptalk-frontline-web-vs-hybrid-vs-native

[9] W3C. "HTML 5.1. A vocabulary and associated APIs for HTML and XHTML" [online]. 2013 [accessed on 19.04.2013]. Available at: http://www.w3.org/TR/html51/

[10] Sergey Mavrody. "Sergey's HTML5 & CSS3 Quick Reference: HTML5, CSS3 and APIs". Third edition. 2012, Belisso. 218 p.

[11] W3C. "Offline Web Applications" [online]. 2013 [accessed on 19.04.2013]. Available at: http://www.w3.org/TR/offline-webapps/

[12] W3C. "Web SQL Database" [online]. 2013 [accessed on 19.04.2013]. Available at: http://dev.w3.org/html5/webdatabase/

[13] "Can I use Web SQL Database?" [online]. 2013 [accessed on 19.04.2013]. Available at: http://caniuse.com/sql-storage

[14] Mozilla. "HTML5 WebSQL for Firefox" [online]. 2013 [accessed on 19.04.2013]. Available at: https://addons.mozilla.org/en-US/firefox/addon/html5-websql-for-firefox/

[15] Wikipedia - Relational database. [online]. 2013 [accessed on 11.11.2013]. Available at: http://en.wikipedia.org/wiki/Relational_database

[16] Sadalage, Pramod J., Fowler, Martin. "NoSQL distilled: A brief guide to the emerging world of polyglot persistence". First edition. 2012, Addison Wesley. 192 p.

[17] Strozzi, Carlo. "NoSQL. A Relational Database Management System" [online]. 1998 [accessed on 19.04.2013]. Available at: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/

[18] Eric Evans's Weblog. [online]. 2009 [accessed on 04.05.2013]. Available at: http://blog.sym-link.com/2009/05/12/nosql_2009.html

[19] NOSQL meetup. [online]. 2009 [accessed on 07.05.2013]. Available at: http://nosql.eventbrite.com/

[20] Couchbase Developer's Guide 2.0. "Comparing Document-Oriented and Relational Data" [online]. 2010 [accessed on 04.05.2013]. Available at: http://www.couchbase.com/docs/couchbase-devguide-2.0/documented-oriented-data-model.html

[21] "DB-Engines Ranking of Key-value stores" [online]. 2013 [accessed on 12.10.2013]. Available at: http://db-engines.com/en/ranking/key-value+store

[22] Redis [online]. 2013 [accessed on 12.10.2013]. Available at: http://redis.io/

[23] memcached [online]. 2013 [accessed on 12.10.2013]. Available at: http://www.memcached.org/

[24] "DB-Engines Ranking of Document Stores" [online]. 2013 [accessed on 12.10.2013]. Available at: http://db-engines.com/en/ranking/document+store

[25] MongoDB [online]. 2013 [accessed on 12.10.2013]. Available at: http://www.mongodb.org/

[26] CouchDB [online]. 2013 [accessed on 12.10.2013]. Available at: http://couchdb.apache.org/

[27] "DB-Engines Ranking of Wide Column Stores" [online]. 2013 [accessed on 12.10.2013]. Available at: http://db-engines.com/en/ranking/wide+column+store

[28] Cassandra [online]. 2013 [accessed on 12.10.2013]. Available at: http://cassandra.apache.org/

[29] HBase [online]. 2013 [accessed on 12.10.2013]. Available at: http://hbase.apache.org/

[30] "DB-Engines Ranking of Graph DBMS" [online]. 2013 [accessed on 03.10.2013]. Available at: http://db-engines.com/en/ranking/graph+dbms

[31] Neo4j [online]. 2013 [accessed on 03.10.2013]. Available at: http://www.neo4j.org/

[32] "Why NoSQL? Three trends disrupting the database status quo" [online] [accessed on 16.10.2013]. Available at: http://www.couchbase.com/why-nosql/nosql-database

[33] Technopedia - Consistency [online] [accessed on 18.10.2013]. Available at: http://www.techopedia.com/definition/27386/consistency-databases

[34] Brewer, Eric A. "Towards Robust Distributed Systems" [online]. 2000 [accessed on 18.10.2013]. Available at: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[35] Nancy Lynch, Seth Gilbert. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services" [online]. 2002 [accessed on 18.10.2013]. Available at: http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf

[36] IrisCouch [online]. 2013 [accessed on 21.10.2013]. Available at: http://www.iriscouch.com/

[37] jQuery Mobile [online]. 2013 [accessed on 21.10.2013]. Available at: http://jquerymobile.com/

[38] ThemeRoller for jQuery Mobile [online]. 2013 [accessed on 09.11.2013]. Available at: http://jquerymobile.com/themeroller/

[39] Jon Fingas. "Android tops 81 percent of smartphone market share in Q3" [online]. Engadget, 31.10.2013 [accessed on 11.11.2013]. Available at: http://www.engadget.com/2013/10/31/strategy-analytics-q3-2013-phone-share/

[40] Android Developers - Notifications [online]. Google [accessed on 11.11.2013]. Available at: http://developer.android.com/guide/topics/ui/notifiers/notifications.html

[41] TouchDB. CouchDB-compatible embeddable database engine for mobile & desktop apps [online]. 2013 [accessed on 09.11.2013]. Available at: http://labs.couchbase.com/TouchDB-iOS/

[42] Couchbase Lite [online]. 2013 [accessed on 08.11.2013]. Available at: http://www.couchbase.com/communities/couchbase-lite

A. LIST OF COUCHDB VIEWS

A.1 Coach database

Design document name	View name
privateNotes	by_customer_id

A.2 Personal database

Design document name	View name
messages	by_date
notifications	next_events
summary	activity_tracker
	constant_hr
	diary
	interval_hr
	just_do_it
	waist
	weight
taskResults	by_date
	by_task_id
tasks	active
	by_date
	by_id
timeline	by_date
trainingProgram	by_id
	by_name

A.3 Program storage

Design document name	View name
trainingProgramTemplate	by_name